Interleaved Bitstream Execution for Multi-Pattern Regex Matching on GPUs

Tianao Ge
The Hong Kong University of Science and Technology (Guangzhou)
Guangzhou, Guangdong, China
tge601@connect.hkust-gz.edu.cn

Xiaowen Chu
The Hong Kong University of Science and Technology (Guangzhou)
Guangzhou, Guangdong, China
xwchu@hkust-gz.edu.cn

Hongyuan Liu*
Stevens Institute of Technology
Hoboken, New Jersey, USA
hliu96@stevens.edu

Abstract

Pattern matching is a key operation in unstructured data analytics, commonly supported by regular expression (regex) engines. Bit-parallel regex engines compile regexes into bitstream programs, which expose fine-grained parallelism and are well-suited for GPU execution. A straightforward strategy executes each bitstream instruction sequentially, processing all data blocks in a loop. However, this execution suffers from poor data reuse and high memory consumption, limiting throughput. Our key insight is to adopt an interleaved execution model, where all bitstream instructions are fused into a single loop and executed block-wise. While interleaved execution could improve data reuse, enabling it on GPUs is non-trivial due to cross-block data dependencies. To address this, we introduce 1) Dependency-Aware Thread-Data Mapping, which resolves cross-block dependencies via selective recomputation. We further improve interleaved execution performance with two additional optimizations: 2) Shift Rebalancing, which balances dependency chains to reduce synchronization barriers; and 3) Zero Block Skipping, which exploits bitstream sparsity to skip computation on zero blocks. Together, these techniques make interleaved execution practical and efficient. Experiments on real-world regex benchmarks demonstrate a 19.5× geometric mean speedup over the state-of-the-art GPU regex engine.

CCS Concepts

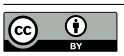
• **Theory of computation** → **Regular languages**; • **Computing methodologies** → *Parallel computing methodologies*; • **Software and its engineering** → Compilers.

Keywords

Regex Matching, Bitstream, GPU, Compiler, Bit Parallelism

ACM Reference Format:
Tianao Ge, Xiaowen Chu, and Hongyuan Liu. 2025. Interleaved Bitstream Execution for Multi-Pattern Regex Matching on GPUs. In *58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25), October 18–22, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3725843.3756052

*Part of this work was done while the author was at HKUST (Guangzhou).



MICRO '25, Seoul, Republic of Korea

ACM ISBN 979-8-4007-1573-0/25/10
https://doi.org/10.1145/3725843.3756052

1 Introduction

Unstructured data analytics frequently involves pattern matching and token extraction tasks such as identifying fields in log entries [46], detecting intrusion patterns in network packets [17, 18, 83], filtering and querying textual data [12, 32, 34–36, 44, 49], genome sequence analysis [11, 21, 23, 58, 65], electronic design automation [27], and preprocessing massive text corpora for language model training [20, 63]. Over 80% of all data generated worldwide is unstructured [42]. Central to these workloads are regular expression (regex) engines, which efficiently express and match complex textual patterns. However, traditional regex engines, which rely on finite automata (e.g., Deterministic Finite Automata, DFAs or Non-deterministic Finite Automata, NFAs) [5, 39], suffer significant performance penalties due to irregular memory accesses and control flows. These issues become even more severe in multi-regex workloads, where thousands of regexes are matched over a shared input stream [28, 37, 88, 89].

To mitigate these inefficiencies, bit-parallel approaches have been proposed. These approaches compile regexes into bitstream programs, where variables are unbounded (potentially very long) bitstreams [24, 50]. Matches are encoded as `1`s in specific positions of these bitstreams. The final match results are computed by executing a sequence of bitstream instructions, which include operations (e.g., `AND`, `OR`, `SHIFT`), and control flow statements (e.g., `if` and `while`). This model exposes fine-grained parallelism over the input stream and avoids the one-byte-at-a-time execution model, making it promising for GPU execution where massive parallelism is essential to fully utilize the hardware. Yet, despite its promise, existing bitstream-based engines are primarily designed for CPUs with SIMD support, and little is known about how to generate high-performance GPU kernels from such programs.

Figure 1 (a) shows a straightforward way to execute bitstream programs on GPUs [24]. To process an instruction that operates on multiple unbounded bitstreams, the bitstreams are divided into blocks. In each step (t_1, t_2, t_3, …), a block of the result bitstream is computed, and the full computation of one bitstream instruction is completed block by block in a loop. The next instruction in the bitstream program is not executed until the current one completes. One critical limitation of this approach is the lack of data reuse: Each bitstream, even if temporary, must be fully materialized and stored in memory, leading to high memory traffic and potentially exceeding GPU memory capacity.

To address this challenge, our key insight is to *interleave the execution of bitstream instructions across blocks*, as shown in Figure 1 (b). Instead of processing each instruction across all blocks before moving to the next instruction, we fuse the entire bitstream

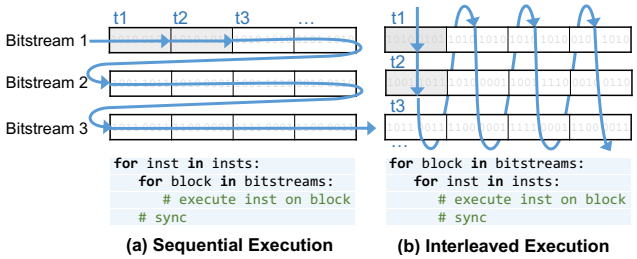


Figure 1: Sequential vs. interleaved execution. Interleaved execution improves data reuse but is hindered by cross-block dependencies and synchronization overhead; our techniques address these challenges to make it practical and efficient.

program into a single loop in a GPU kernel and execute all instructions on the same input block. Interleaved execution improves data reuse and reduces memory pressure. However, enabling correct and efficient interleaved execution is non-trivial. Shift operations create cross-block dependencies, as they may require results from "past" (right shifts) or "future" (left shifts) blocks in the bitstream. This becomes more complex with control flow constructs (e.g., `while` loops) that introduce dynamic, input-dependent execution paths.

We propose BitGen, a code generator that enables *efficient interleaved execution* through three key techniques: 1) To enable interleaved execution, we introduce **Dependency-Aware Thread-Data Mapping**, which resolves inter-block data dependencies by assigning GPU threads to blocks of bitstreams dynamically and selectively *recomputing* only the bits necessary for the current block. This enables fine-grained reuse of intermediate bitstreams at the cost of per-block barriers introduced to satisfy bitstream instruction dependencies. 2) To alleviate the synchronization overhead introduced by interleaved execution, we propose **Shift Rebalancing**. It transforms long dependency chains involving shift operations (e.g., `SHIFT` followed by `AND`) into a balanced and shallower dataflow graph. This enables better instruction scheduling and allows independent instructions to share synchronization points, reducing the total number of barriers. 3) Finally, interleaved execution exposes new optimization opportunities. Many intermediate bitstreams are sparse, often consisting of mostly zero blocks. Unlike in sequential execution, where all blocks are processed regardless of content, interleaved execution allows us to exploit this sparsity. We introduce **Zero Block Skipping**, which inserts compile-time `if` conditions to skip computation for zero blocks, thereby mitigating redundant computation.

To the best of our knowledge, BitGen is the *first work that compiles entire bitstream programs into a GPU kernel that enables interleaved execution*. We make the following contributions:

- We identify key performance limitations of sequential block-wise execution on GPUs, including poor data reuse, high memory consumption, and redundant computation.
- We propose BitGen, a GPU code generator that enables efficient interleaved execution of bitstream programs through three key optimizations: Dependency-Aware Thread-Data Mapping for resolving cross-block dependencies via selective recomputation, Shift Rebalancing for reducing synchronization barriers through dependency balancing, and Zero Block Skipping for skipping computation on sparse bitstreams.
- We evaluate BitGen on real-world regex workloads, demonstrating 19.5× and 1.7× geometric mean speedup over state-of-the-art GPU and CPU regex engines, respectively.

Listing 1: Simplified Grammar of Regular Expressions

```
R ::= CC      // character class (e.g., a, [a-z])
    | RR      // concatenation
    | R|R     // alternation
    | R*      // kleene star
    | R{n,m}  // bounded repetition
    | R+      // one or more repetitions
    | R?      // zero or one repetition
```

2 Background

This section introduces the basics of regular expressions (regexes), and demonstrates how regexes are translated to bitstream programs.

Regular Expression. Regular expressions (regex) are a compact and expressive way to represent patterns. They are widely used in tasks such as text processing, network intrusion detection, bioinformatics, and parsing. For example, the network intrusion detection systems use regexes to represent attack signatures and match malicious payload patterns in real-time packet streams [7, 64, 91]. Listing 1 defines a basic grammar for regular expressions.

A *character class* CC matches a single character, such as '`a`' or characters represented as '`[a-z]`'. Concatenation (RR) matches when one pattern is followed directly by another. Alternation ($R|R$) allows either of the two patterns to match. The Kleene star ($R*$) accepts zero or more repetitions of a pattern. Finally, repetition operators include $R\{n, m\}$ for bounded repetition, allowing between n and m matches; $R+$ for one or more matches; and $R?$ for zero or one match. For example, `[a-z0-9]+@[a-z0-9]+\.[a-z]{2,}` matches an email address where both the username and domain consist only of lowercase letters and digits.

Regular expressions and finite automata are mathematically equivalent: any pattern described by a regex can be recognized by a finite automaton, and any language accepted by a finite automaton can be described by a regex [10, 75]. This equivalence allows regex engines to compile regexes into automata such as non-deterministic or deterministic finite automata (NFAs or DFAs), and perform matching by simulating state transitions one symbol at a time [5, 16, 17, 29, 40, 84]. However, simulating automata on GPUs introduces challenges due to irregular memory accesses, control flow divergence, and limited parallelism, which can significantly hinder performance on massively parallel architectures [37, 52, 55, 77].

Bitstream Program. To address the inefficiencies of automata on modern parallel hardware, alternative approaches based on bit-parallelism have been proposed [24, 50]. The key idea of this approach is to lower regular expressions into data-parallel **bitstream programs** (Listing 2), a sequence of operations over bitstreams using Boolean logic and control flow. Each statement either assigns an expression to a variable or implements control flow (e.g., `if` or

Listing 2: Grammar of Bitstream Program

```
<stmts> ::= <stmt> <stmts> | ε
<stmt>  ::= <var> "=" <expr> | "if" "(" <var> "):" <stmts>
          | "while" "(" <var> "):" <stmts>
          | "for" "(" <const> ")" ":" <stmts>
<expr>  ::= <op> <var> | <var> <op> <var>
          | <var> <shift> <const>
<op>    ::= "~" | "&" | "|"          # NOT, AND, OR
<shift> ::= "≪" | "≫"                # LSHIFT, RSHIFT
```

```
# input: text, CC
# output: S_cc
b = transpose(text)
S_cc = match(b, CC)
```

(a) Character class

```
# input: S1, S2
# output: S_R1R2
S1 = S1 ≫ 1
S_R1R2 = S1 & S2
```

(b) Concatenation: R_1R_2

```
# input: S1, S2
# output: S_R1|R2
S_R1|R2 = S1 | S2
```

(c) Alternation: $R_1|R_2$

```
# input: S1
# output: S_R{n,m}
S2 = S1
for i = 1 to n - 1:
    S2 = S2 ≫ 1
    S2 = S2 & S1
S_R{n,m} = S2
for i = 1 to m - n:
    S2 = S2 ≫ 1
    S2 = S2 & S1
    S_R{n,m} = S_R{n,m} | S2
```

(d) Bounded repetition: $R\{n,m\}$

```
# input: S1, S2
# output: S_R1R2*
S_R1R2* = S1
while (S1):
    S3 = S1 ≫ 1
    S4 = S2 & S3
    S5 = ~S_R1R2*
    S1 = S4 & S5
    S_R1R2* = S_R1R2* | S4
```

(e) Kleene star: $R_1(R_2)*$

Figure 2: Rules for lowering regexes to bitstream programs, omitting R+ and R? as derivable from existing constructs. (b): Concatenation is applied at the character-class step. Here, R_2 is a single character class. (e): R_1 is used as a prefix to determine the starting positions in the Kleene star, since R_2* alone would mark all positions as matches.

while). In control-flow statements, the condition is a bitstream that evaluates to true if it contains at least one set bit (i.e., popcount > 0). Expressions operate over bitstreams and include unary or binary bitwise operations, as well as shift operations with immediate constants. We refer to a statement applying bitwise, shift, or conditional operations on bitstreams as a **bitstream instruction**.

Lowering Regular Expressions to Bitstream Programs. Figure 2 illustrates the rules for this compilation process. For a regex R, we define the bitstream S_R such that a `1` at position i indicates a successful match ending at that position. For example, if R is `/cat/` and the input stream is bob**cat**, then S_{cat} = `000001`. We adopt all-match semantics [24, 62], in which all possible match endpoints are preserved during execution (e.g., every position matching `a*` is marked). As regexes are structured as compositions of subexpressions, S_R could be computed by the bitstreams of R's subregexes.

To facilitate bit-parallel processing, the input byte stream is transposed into 8 bitstreams (b_0 to b_7) [24, 50]. Each bitstream b_i represents the i-th bit of all bytes in the input. This representation allows character-class matches to be computed using parallel bitwise operations. For example, the bitstream that matches 'a' (i.e., ASCII `01100001`) is computed as: $\neg b_0 \wedge b_1 \wedge b_2 \wedge \neg b_3 \wedge \neg b_4 \wedge \neg b_5 \wedge \neg b_6 \wedge b_7$. Figure 2 (a) illustrates this process, where `text_trans` corresponds to bitstreams b_0 to b_7, and CC is the bitmask of the character class (e.g., `01100001` for 'a'). The resulting bitstream (S_{CC}) marks all matching positions with a `1` and all others with a `0`.

To match a **concatenation** R_1R_2, as shown in Figure 2 (b), the bitstream S1 can be viewed as a set of cursors, where each `1` marks a position in the input stream where R_1 finishes. Here, R2 is a single character class. By shifting S1 one position to the right, we advance these cursors and begin matching R_2 from the next position. For a general R_1R_2, we first decompose R_2 into $CC_1 \cdots CC_L$ and apply this rule L times; e.g., abc · de is evaluated as $(((abc \cdot d) \cdot e))$ with two successive one-position shifts. The bitwise AND operation with S2 identifies the positions where both R_1 and R_2 match in sequence, thereby producing S_{R1R2}. **Alternation** $R_1 \mid R_2$ is handled by taking the union of the two bitstreams, i.e., $S_{R1|R2}$ = S1 | S2, as shown in Figure 2 (c). Figure 2 (d) illustrates the program for computing **bounded repetition** of a regex R. The code uses RSHIFTs and ANDs to ensure each repetition of R follows the previous one. By chaining these operations, it constructs match streams for $S_{R\{n,n\}}$, $S_{R\{n+1,n+1\}}$, …, $S_{R\{m,m\}}$, and combines them using bitwise OR to produce the final result $S_{R\{n,m\}}$. Figure 2 (e) shows how to compute the **Kleene star** of R_2, with the starting positions provided by the matching results of R_1. It iteratively accumulates all positions that can be reached through repeated applications of R_2. The process is expressed as a fixed-point loop: It repeatedly shifts the matching positions forward, finds matches of R_2, and stops when no new matches are found.

Listing 3: Bitstream program for regex `/a(bc)*d/`

```
# input: text
# output: S12
S1, S2, S3, S4 = match(text_trans, CCs) # a, b, c, d
S10 = S1
while (S1):
    S5 = S1 ≫ 1
    S6 = S2 & S5
    S7 = S6 ≫ 1
    S8 = S3 & S7
    S9 = ~S10
    S1 = S8 & S9
    S10 = S10 | S8                    # a(bc)*
S11 = S10 ≫ 1
S12 = S4 & S11                        # a(bc)*d
```

Example: lowering `/a(bc)*d/` to a bitstream program. As shown in Listing 3, the initial match of a produces S1, which serves as the starting positions for the Kleene star (bc)*. The star body is decomposed at compile time into two character classes, CC_b and CC_c. At runtime, the fixed-point loop repeatedly applies the two shifts and AND operations to compute all positions reached by (bc)*, accumulating results in S10. Finally, S10 is shifted by 1 and ANDed with the match of CC_d to produce S12.

```
# input: text. output: S9
S1, S2, S3, S4 =
     match(text_trans, CCs)
S5 = S1 ≫ 1
S6 = S5 & S2          # ab
S8 = 0
if (S6):
    S7 = S6 ≫ 1
    S8 = S7 & S3      # abc
S9 = S8 | S4          # abc|d
```

(a) Bitstream program for regex `/(abc)|d/`.

Text	a	b	c	d	a	b	c	e
S1	1	.	.	.	1	.	.	.
S2	.	1	.	.	.	1	.	.
S3	.	.	1	.	.	.	1	.
S4	.	.	.	1	.	.	.	.
S5	.	1	.	.	.	1	.	.
S6	.	1	.	.	.	1	.	.
S7	.	.	1	.	.	.	1	.
S8	.	.	1	.	.	.	1	.
S9	.	.	1	1	.	.	1	.

(b) Bitstream program execution illustrated step by step. Zeros are shown as dots.

Figure 3: Example of bitstream program and its bitstream. S1 to S4 are bitstreams for character classes a, b, c, and d. The transpose of the input text and the computation of these character classes are omitted in this figure.

Runtime example of `/(abc)|d/`. Figure 3 presents a step-by-step example of matching the regex `/(abc)|d/` against the input stream abcdabce. For improved readability, zeros are displayed as dots. The matching positions are marked with `1` in S9. Although the code generation rules in Figure 2 do not generate `if` clauses, adding them can be beneficial: in Figure 3, when S6 is all `0`, both S7 and S8 will also be `0`, allowing us to skip several operations. We will discuss this in detail in Section 6.

3 BitGen: Generating Efficient GPU Kernels for Bitstream Programs

3.1 Overview of BitGen

We propose BitGen, a GPU kernel code generator for bitstream programs. Figure 4 illustrates the compilation and execution workflow of BitGen. This work targets the problem of *multi-regex matching*, where a set of regular expressions are evaluated concurrently over a shared input stream–a critical need in high-throughput domains such as deep packet inspection and log analytics [7, 64, 83, 91]. We use Parabix [50] to compile regexes into bitstream programs. Unlike Parabix, which lowers these programs into CPU SIMD instructions, BitGen emits optimized GPU kernels for execution.

Analogous to several domain-specific pattern matching accelerators [31, 68, 73], the generated kernel follows an MISD-style execution model at runtime for the bitstream programs: each GPU Cooperative Thread Array (CTA) executes a sequence of instructions of a regex-derived bitstream program while consuming a shared input stream. When multiple input streams are processed concurrently, the model transitions to MIMD-style execution.

To leverage regex-level parallelism on GPUs, BitGen partitions the regexes into groups and generates a bitstream program for each group. Each bitstream program is then assigned as a device function to a CTA. We describe the regex grouping strategy in Section 7. Within a CTA, threads execute the bitstream instructions in a data-parallel manner. To process long bitstreams, computation is divided into multiple iterations. Each bitstream (S) is partitioned into blocks (B), and each loop iteration processes one block (i.e., Block-wise Execution). Threads collectively load the block's data

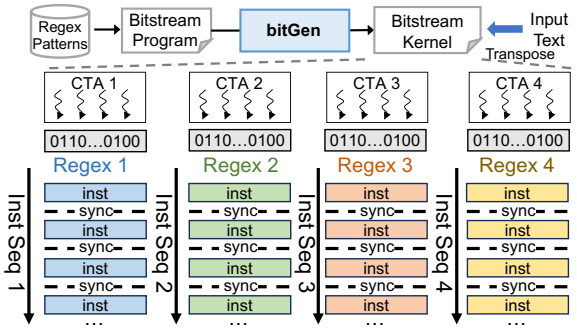


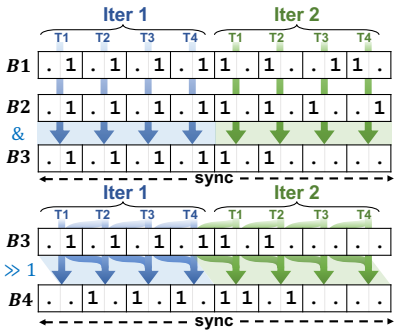
Figure 4: BitGen workflow: Bitstream programs are compiled into a GPU kernel, with each CTA processing one or more regexes.

```
S3 = S1 & S2
S4 = S3 ≫ 1
```

(a) Bitstream Program.

```
for Bi in S:
  B3i = B1i & B2i
for Bi-1,Bi in S:
  B4i = [B3i-1, B3i] ≫ 1
```

(b) Block-wise Sequential Execution in CTAs.



(c) Thread Data Mapping. $T = 4$, $W = 2$.

Figure 5: Example of the sequential execution of a bitstream program. In (b), we denote S as the bitstream and B_i as the block from the bitstream processed in the i-th iteration. The notation $[B_{i-1}, B_i] \gg 1$ represents concatenating two consecutive blocks, right shifting, and extracting the higher bits as the new block. Block writes to global memory and synchronizations are omitted in (b).

from global memory into registers and compute in parallel. Each thread handles a unit (U) of data per iteration, where the unit size (W) is determined by the GPU word size (typically 32 bits). The number of blocks is computed as $N = \lceil |S|/(T \cdot W) \rceil$, where $|S|$ is the number of bits in S, and T is the number of threads per CTA. For clarity, we represent the bitstream as $S = \{B_0, B_1, \ldots, B_{N-1}\}$.

3.2 Sequential Block-wise Execution

Figure 5 presents an illustrative example of the sequential execution of a bitstream program. The bitstream program shown in Figure 5 (a) is compiled into a device function consisting of two loops (Figure 5 (b)). Each loop (represented as a dashed blue box, a convention used in later figures) corresponds to a bitstream instruction and iterates over the blocks of the operand bitstreams. For the first bitstream instruction, which performs a bitwise AND, each thread processes its assigned units independently from $B1_i$ and $B2_i$. In contrast, the second instruction performs a right-shift

operation across block boundaries. To correctly shift bits across blocks, the loop must load two adjacent blocks ($B3_i$ and $B3_{i-1}$), enabling threads to access bits from the previous block when computing the current one. Figure 5 (c) illustrates the thread-to-data mapping for this process, assuming $T = 4$ and $W = 2$. Due to instruction dependencies, an intra-CTA *barrier* is inserted between loops. However, sequential block-wise execution presents several **performance challenges**:

a) Poor Data Reuse. In sequential block-wise execution, each instruction executes in a dedicated loop, causing frequent loading and storing of intermediate bitstreams. This loop separation severely limits data reuse opportunities, resulting in poor cache utilization and unnecessary data movement.

b) High Memory Consumption from Intermediate Bitstreams. Separate loops per instruction require intermediate bitstreams to be explicitly stored between loops. As programs grow, these temporary bitstreams significantly increase GPU memory usage (further analyzed in Table 4), limiting scalability.

c) Missed Opportunities to Skip Redundant Computation. Intermediate bitstreams often contain large portions of zeros due to partial or complete regex mismatches, which occur commonly in practice [53, 78]. However, loops in sequential execution lack visibility into subsequent computations, making it difficult to dynamically detect and skip these redundant blocks.

3.3 Key Insight: Interleaved Execution

Our key insight is to adopt an *interleaved execution* model, in which all bitstream instructions are fused into a single loop. In each iteration, the fused loop executes *all bitstream instructions* on the current data block before proceeding to the next, enabling better data reuse, lower memory consumption, and the ability to skip redundant computation. To realize it, this paper presents three key techniques (Section 4, Section 5, and Section 6). Together, they are integrated into BitGen, enabling interleaved execution to be both practical and performant for real-world bitstream workloads.

4 Enabling Interleaved Execution via Dependency-Aware Thread-Data Mapping

This section describes the cross-block dependencies in interleaved execution and how we resolve them.

4.1 Challenges of Interleaved Execution

Figure 6 (a) shows two bitstream programs and their block-wise execution using a straightforward GPU kernel, also shown as sequential execution in Figure 1 (a). Each instruction runs in a separate loop that iterates over its operands, writing results back to global memory before the next instruction begins.

To improve data reuse, we propose interleaved execution via loop fusion. Operand blocks are loaded once, and intermediate results are kept in registers across instructions, reducing global memory traffic. Basic instructions like `AND` and `OR` are easy to fuse since they use only thread-local data, but care is needed for instructions with cross-block dependencies:

Cross-block Dependencies. Figure 6 (b) shows the loop-fused execution of the two example programs. In Example 1, the `AND` instruction is fused with the `SHIFT` instruction. Each CTA consists of 4 threads ($T = 4$), with each thread processing a data unit of 2 bits ($W = 2$). The bitstream consists of two blocks ($N = 2$). Example 1 illustrates a data dependency violation caused by the `SHIFT` operation under interleaved execution. The `SHIFT` operation at line 2 depends on the result of the `AND` operation at line 1. Since each thread holds its unit of `B3` in registers, it accesses the preceding thread's data via shared memory within the CTA. However, since the fused kernel executes block-wise across two iterations, the first thread in the second iteration cannot access the last bit of the preceding block's result (`B3`), which is required for the `RSHIFT` operation. As a result, the first bit of `B3` in iteration 2 is missing, leading to incorrect output.

One possible solution to address cross-iteration dependencies is to forward intermediate results via registers. However, this approach is not generally applicable due to several limitations. First, when `SHIFT` instructions appear frequently, each would require a dedicated register for forwarding. Second, `SHIFT` instructions in bitstream programs may appear in both directions (i.e., left and right). Register forwarding inherently supports only one direction (e.g., from the previous iteration), and thus cannot handle bi-directional dependencies. Third, cross-block dependencies caused by `SHIFT` also affect the correctness of control-flow constructs:

In Example 2, the program includes a conditional statement that checks whether `S1` is all zeros. When regular expressions are compiled into bitstream programs, such control flow constructs are transformed into predicated execution, where all the instructions inside a conditional block are guarded by a predicate derived from the condition variable (e.g., whether `S1` contains any `1`s). While predicated instructions simplify analysis and enable loop fusion across conditional blocks, they do not eliminate data dependencies across iterations in interleaved execution: Example 2 in Figure 6 (b) shows that in Iteration 2, since `B1` is all zeros, the loop chooses to skip the instructions inside the `if` block. However, this decision is incorrect: the `SHIFT` operation in line 3 still propagates a bit from the previous iteration, leading to an incorrect value in $B4_2$.

Overall, shift operations introduce cross-block dependencies that pose correctness challenges for interleaved execution. We address these challenges with a dependency-aware thread-data mapping strategy, as detailed in Section 4.2.

4.2 Dependency-Aware Thread-Data Mapping

We introduce a dependency-aware thread-data mapping scheme that addresses the cross-block dependencies shown in Section 4.1. The key idea of our approach is to *recompute the required bits within the current iteration, avoiding the complexity and limitations of forwarding intermediate results from "past" or "future" iterations.*

Illustrative Example of Dependency-Aware Mapping. To determine which bits we should recompute for each block, we begin with a simple case (Example 1 in Figure 6) where the shift distance is statically known at compile time. In Figure 6 (c), the `RSHIFT` operation at line 2 shifts by one bit, which introduces a cross-block dependency: the first bit in the current iteration depends on the last bit of the previous iteration. To ensure correctness, we adjust the starting position of each iteration so that it overlaps with the preceding block by enough bits to cover the shift distance. Given

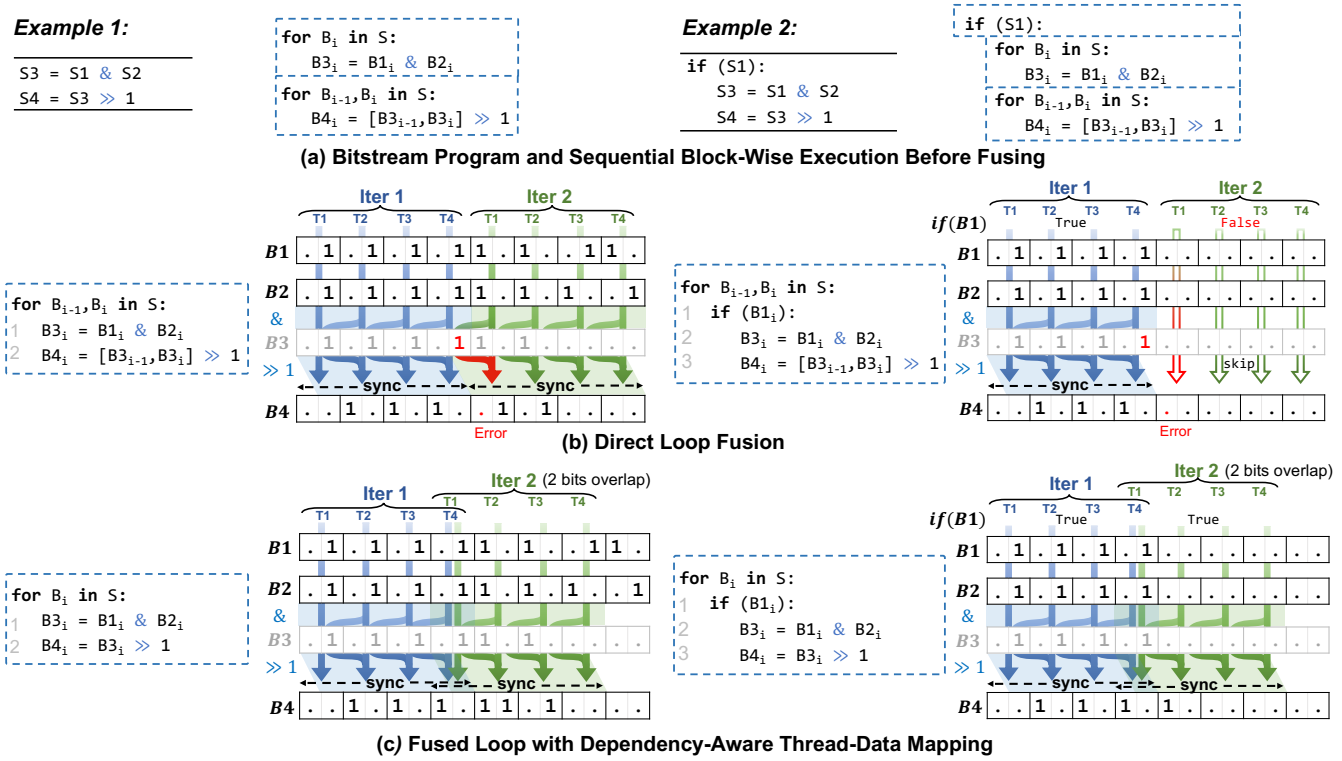


Figure 6: Challenges in enabling interleaved execution via loop fusion. (a) Original bitstream program with sequential block-wise execution. (b) Directly fusing all bitstream instructions into one loop introduces correctness issues due to cross-block dependencies. (c) Our approach resolves these dependencies by recomputing necessary bits through dynamic thread-to-data mapping. Intermediate blocks stored only in thread-local registers are shown in gray.

a word size W, CTA size T, and the number of *overlap distance* in bits Δ, we compute the iteration offset as $T - \lceil \Delta/W \rceil \times W$. This ensures that all bits required by the SHIFT operation are recomputed within the current iteration. For example, when $W = 2$ and $\Delta = 1$, the second iteration starts at bit position 6 instead of 8, recomputing the necessary bits (the 7th bit) to maintain correctness. This mapping ensures correctness for both left and right shift operations by overlapping words just enough to cover any cross-block dependencies introduced by bitwise shifts in either direction.

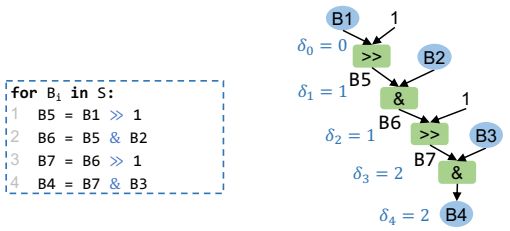
Computing Overlap Distance. We consider the dataflow graph (DFG) of the bitstream program. We define δ_i as the cumulative bit offset introduced by SHIFT operations up to the i-th operation along a dataflow path. Starting from $\delta_0 = 0$, each SHIFT instruction updates the offset as: $\delta_{i+1} = \delta_i + k_i$, where k_i is the signed shift distance at step i (positive for right shift, negative for left shift). This sequence $\{\delta_i\}$ reflects the positions of all accessed bits along the path P. We define the required overlap distance as:

$$\Delta = \max_{P \in \text{Paths}} \left(\max_i \delta_i - \min_i \delta_i \right)$$

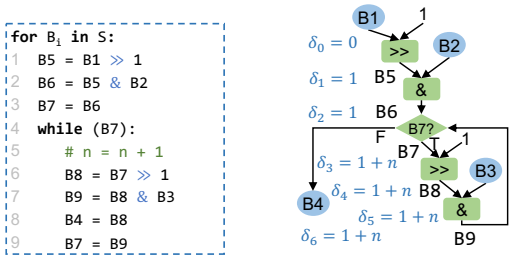
Figure 7 (a) shows an example with two right shifts. In the path B1 → B5 → B6 → B7 → B4, each right shift accumulates a bit offset from the original input B1. With two shifts along the path, the combined effect results in accessing a bit two positions earlier in B1 when computing each bit in B4 ($\Delta = 2$). We illustrate another example to clarify the sequence $\{\delta_i\}$. Consider a bitstream program with one dataflow path of two operations: first a right shift by 1, and then a left shift by 2 (b = a ≫ 1, c = b ≪ 2). The resulting cumulative offset sequence is $\{0, 1, -1\}$. To ensure correctness, the current block starts 2 bits earlier and recomputes them ($\Delta = 2$). This covers the 1 bit required by the previous block due to the right shift and the 2 left-shifted bits within the current block.

Tracking the cumulative shift offset δ_i at each step allows us to precisely determine which input bits are accessed. This is essential for computing the overlap distance Δ, which guarantees correctness under interleaved execution. After Dependency-Aware Mapping, the fused loop requires three iterations to compute in example 1 of Figure 6 (c). We will discuss the overhead of recomputing in Section 8.2.

Handling Loops. When the DFG contains multiple loops (either nested or parallel), we need to model how SHIFT operations accumulate across different loop iterations. To handle this, we express the overlap distance Δ as a function of all loop iteration counters $n_1, n_2, ..., n_L$, , where L is the number of loops in the DFG. For each shift instruction s, we define a *multiplicity function* $\mu_s(n_1, ..., n_L)$, which represents how many times this shift instruction is executed, depending on the loop structure it belongs to. The cumulative bit offset δ_i at each step along a path is then: $\delta_{i+1} = \delta_i + \mu_s(n_1, ..., n_L) \cdot k_s$.



(a) The loop with fixed overlap distance ($\Delta = 2$).



(b) The loop with overlap distance offset ($\Delta = 1 + n$, where n is the loop counter determined at runtime).

Figure 7: Illustration of overlap distance computation in bitstream programs. Left: Bit-wise execution of the bitstream program. Right: Dataflow graph with cumulative bit offsets.

The overall overlap requirement is given by:

$$\Delta(n_1, ..., n_L) = \max_{P \in \text{Paths}} \left(\max_i \delta_i - \min_i \delta_i \right)$$

This formulation generalizes to loops, enabling dynamic overlap computation under interleaved execution. For example, Figure 7 (b) shows a program containing a `while` loop along with its corresponding DFG. In this program, the dataflow path from input `B1` to output `B4` includes two shift operations: the first `RSHIFT` on `B1` occurs outside the loop and is executed once, while the second `RSHIFT` on `B7` resides inside the `while` loop and is executed once per loop iteration. Assuming the loop executes n times, the cumulative bit offset along the path becomes: $\delta(n) = 1 \cdot (+1) + n \cdot (+1) = 1 + n$. This means that computing `B4` requires accessing bits in `B1` that are $1 + n$ positions earlier, which may reside in a different interleaved iteration. To ensure correctness, the current iteration must be able to recompute all bits within this range, and the required overlap becomes $\Delta(n) = 1 + n$. Thus, the loop iteration counter n will be inserted into the program to record the required overlap distance for each iteration.

5 Reducing Synchronization Overhead via Shift Rebalancing

This section describes how interleaved execution introduces additional synchronization and how we reduce it.

5.1 Source of Synchronization

This section discusses the synchronization challenges in interleaved execution: 1) Fused loops require intra-loop barriers to resolve cross-block dependencies, and 2) long dependency chains across instructions further exacerbate the overhead.

Intra-Loop Synchronization in Interleaved Execution. In sequential execution, each bitstream instruction is compiled into its own loop. Instructions run one after another, writing results to global memory before the next begins. Since each thread processes independent data and there are no cross-thread dependencies within a loop, no intra-loop synchronization is needed. Interleaved execution, in contrast, fuses all instructions into a single loop to enhance memory reuse and reduce global memory traffic. Intermediate results stay in thread-local registers, avoiding global memory altogether. However, a `SHIFT` operation requires two barriers: one before to ensure all inputs are loaded into shared memory, and one after to ensure outputs are ready for the next bitstream instruction.

Long Dependency Chains. The `SHIFT` operation is a fundamental building block to express concatenation in regular expressions (Figure 2 (b)). Thus, synchronization is introduced whenever concatenation appears in a regex. For example, the left side of Figure 8 shows the interleaved execution code and its dataflow graph for regex `/abb/`. In the DFG, the original program has two `RSHIFT` instructions and two `AND` instructions, forming a dependency chain. Such a dependency chain leaves no room for instruction scheduling. Specifically, the `AND` operation at line 4 depends on `B6`, which becomes available only after the synchronization following the `RSHIFT` at line 3, even though `B3` is already ready for use. We call this structure *unbalanced*, as each instruction depends on the previous, forming a sequential dataflow path. Breaking the dependency chain and making `SHIFT` instructions schedulable allows multiple `SHIFT` operations to share common synchronization points, offering the opportunity to reduce synchronization overhead.

5.2 Rebalancing Dependency Chains

This section presents our technique for breaking long dependency chains and restructuring them into more balanced trees.

Operand Rewriting. Long dependency chains with `SHIFT` and bitwise operations can stall execution due to frequent barriers. To reduce this, we apply *operand rewriting*, an algebraic transformation that shifts dependencies between operands. For instance, `(A ≫ n) & B` can be rewritten as `(A & (B ≪ n)) ≫ n`, assuming unbounded bitstreams. This preserves semantics while removing `SHIFT` from the critical path, allowing earlier scheduling for the ready operand. In this example, if B is ready first, rewriting allows the `SHIFT` on B to execute earlier.

Shift Rebalancing for Bitstream Programs. Operand rewriting ensures the semantic correctness of switching dependencies between operands. To break the long dependency chain in bitstream programs, we apply a transformation pass that uses operand rewriting to rebalance `SHIFT` instructions in the dataflow graph repeatedly. The key idea is to rewrite data dependencies in the graph such that `SHIFT` instructions are applied to operands that appear earlier in the execution order. For example, consider an expression of the form `(A ≫ 1) & B`, where operand A depends on x preceding operations and B depends on y operations in the dataflow graph. If $x > y$, it is beneficial to move the `SHIFT` instructions to B, resulting in `(A & (B ≪ 1)) ≫ 1`. This rewrite reduces the critical path length

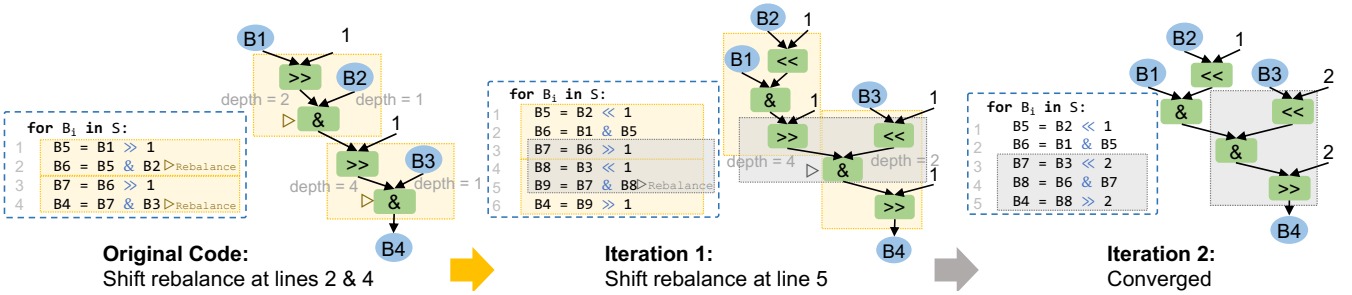


Figure 8: Shift Rebalancing iteratively rewrites operands to produce a more balanced and schedulable DFG.

by shifting the dependency to a shallower operand with fewer preceding operations. Our algorithm performs a topological traversal of the dataflow graph for each iteration. For each AND, if an operand is a SHIFT, it is moved to the operand with lower topological depth, thereby shortening the dependency chain. This transformation is applied iteratively until a fixpoint is reached. An additional iteration is triggered when new opportunities for operand rewriting are identified. Figure 8 shows an example of this process. In the initial graph, two AND instructions (lines 2 and 4) are rebalanced in the first iteration. In the second iteration, the AND at line 5 is further transformed, resulting in a final dataflow graph where the SHIFT instructions apply to B1 and B3. Since B3 is available, the shifted value can be computed earlier. Although this transformation may introduce new SHIFT instructions, they are merged after the last AND instructions (e.g., as seen in line 5 of Iteration 2). Shift Rebalancing improves schedulability, which we leverage to reduce barriers in Section 5.3.

5.3 Merging Barriers of Shift Instructions

This section describes how we schedule and merge the barriers of SHIFT instructions to improve efficiency after Shift Rebalancing.

Scheduling and Merging Barriers. After applying Shift Rebalancing, we reduce synchronization overhead by scheduling and merging SHIFT instructions. A SHIFT can be delayed and scheduled at the point where all its operands become available. Our merging algorithm uses a greedy approach, processing instructions sequentially. For each SHIFT instruction, we attempt to merge it with the preceding SHIFT if: (1) its operands are ready at the position of the preceding SHIFT, and (2) the number of merged SHIFT instructions does not exceed the allowed maximum. The maximum limit is constrained by available shared memory, as merging more instructions requires storing more blocks. When merged, the later instruction is scheduled to the position of the earlier one, allowing their barriers to be combined. If merging is not possible, the current SHIFT instruction becomes the starting point for subsequent merging attempts. In our evaluation, the maximum number of SHIFT instructions that can be merged together (referred to as *merge size*) is exposed as a tunable parameter described in Section 7.

Figure 9 shows an example based on the transformed results from Figure 8. In ❶, the SHIFT instructions at lines 1 and 3 operate directly on input bitstreams B2 and B3, and can therefore be scheduled earlier, near the beginning of the program (❷). Once co-located, the barriers of SHIFT instructions at lines 1 and 2 in ❷ can be

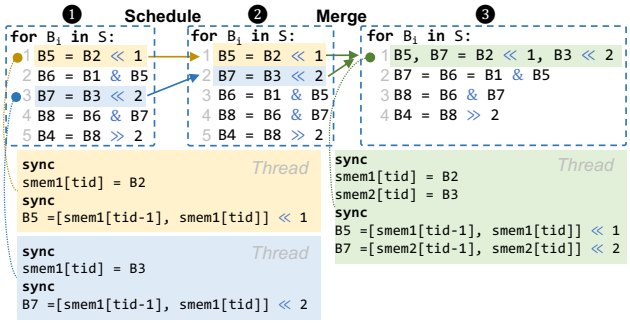


Figure 9: Independent SHIFT operations are scheduled together, allowing their barriers to be merged.

merged. Therefore, B2 and B3 can be written to shared memory with one barrier (❸). Figure 9 also shows per-thread execution before and after merging. Two SHIFT instructions originally require four synchronizations, which are reduced to two.

Removing Redundant Bitstream Copies in Shared Memory. In bitstream programs, multiple SHIFT operations may apply different shift amounts to the same bitstream. For example, when Figure 9 represents the regex /abb/, the same bitstream for character b (B2 and B3) is used in two SHIFT operations—by 1 and 2 bits—to produce B5 and B7. When such operations are merged, we can identify the shared input and avoid storing multiple copies by storing only the unshifted version. The shifted values are then recomputed locally as needed. As more SHIFT operations are merged, this optimization becomes increasingly effective in reducing memory traffic and helps amortize the additional shared memory usage introduced by scheduling and merging barriers.

6 Reducing Redundant Computation through Zero Block Skipping

This section discusses the missed opportunity to skip redundant computation in sequential execution and demonstrates how we exploit it in our interleaved execution.

Zero values are prevalent in bitstreams, especially in regex matching, where a zero bit typically indicates a mismatch. When such mismatches occur, the matching process can terminate early, rendering subsequent operations unnecessary. In bitstream programs, this behavior provides an opportunity for skipping computations that

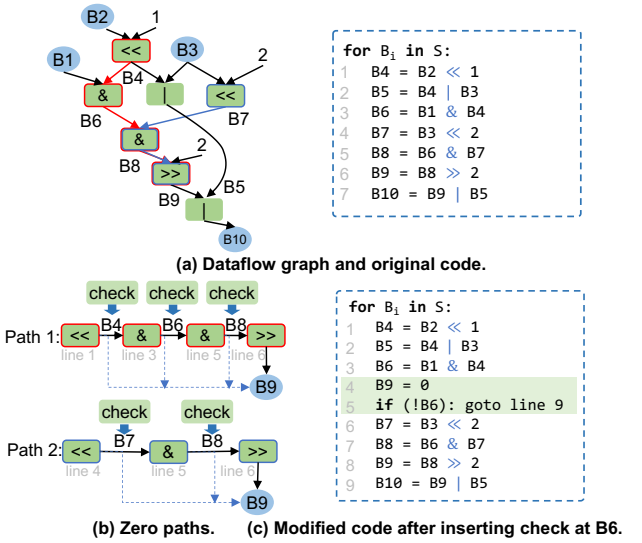


Figure 10: Illustration of identifying zero paths and inserting conditional branches.

are guaranteed to yield zero values, similar to short-circuiting in logical operations. Specifically, operations such as `AND` and `SHIFT` on zero-valued inputs will always yield zero outputs. However, enabling zero-skipping in sequential block-wise execution of bitstream programs on GPUs is challenging. Even if a block is entirely zero, the loop must proceed to the next block, since cross-block dependencies prevent early termination.

Leveraging Interleaved Execution to Exploit Sparsity. To address this limitation, BitGen introduces Dependency-Aware Thread-Data Mapping to enable interleaved block-wise execution by dividing the bitstream into blocks and recompute the necessary bits. All bitstream instructions, including control flow (e.g., Example 2 in Figure 6), are fused into one loop, enabling us to skip redundant work on sparse bitstreams.

Identifying Zero Paths. To exploit sparsity in bitstream execution, we insert runtime conditionals that dynamically skip redundant computations. The key to this optimization is the identification and utilization of *zero paths*. A zero path is a sequence of instructions within the DFG where zero inputs guarantee zero outputs, consisting of zero-preserving operations `AND` and `SHIFT`. At runtime, detecting whether a variable in its zero path is zero allows skipping the remaining chain of computations along this path. Initially, we identify zero paths by traversing the DFG and locating paths composed exclusively of operations that preserve the zero-value property. Figure 10 (a) illustrates a bitstream program along with its DFG. We identify its two zero paths shown in Figure 10 (b).

Conditional Branch Insertion. Once zero paths are identified, we attempt to insert conditional branches to skip unnecessary computations. To reduce the number of conditional branch instructions at runtime, we do not use variable guards for each bitstream instruction. Instead, we insert `goto` statements in CUDA. Specifically, at the head of each zero path, we insert a runtime check (e.g., `if (!B) goto LABEL`), which directly skips to the instruction following the last node in the zero path if `B` is zero. However, since zero paths identified in the DFG might not correspond to contiguous instructions in the linear execution order, simply inserting a guard at the head and skipping directly to the end may inadvertently skip over essential instructions not part of the zero path, potentially causing incorrect results. To ensure correctness, each proposed conditional `goto` along the zero path is validated before insertion. If the skipped range includes an instruction that defines a variable used outside the path, the insertion is rejected. Although instruction reordering could resolve this, we avoid it for simplicity. The process continues at the next node until a valid insertion is found or the path ends. This branch does not cause warp divergence, as all threads in the CTA use a shared condition computed via a block-wide reduction with `atomicOr`.

For example, in Figure 10 (b), we begin by attempting to insert a guard at `B4` in Path 1, the head of a zero path, and it will skip the instructions from line 2 to line 6 in the original code. However, we found that the instruction at line 2 is not on the zero path and is used at line 7, so the insertion fails. Figure 10 (c) shows the modified code after inserting conditional checks at `B6`.

Interval-Based Multi-Guard Insertion. As zero paths can be lengthy, using only one guard at the head may not fully exploit available sparsity. Hence, we introduce an interval parameter I that controls guard insertion frequency along zero paths. Specifically, every I instructions along a zero path, we attempt the insertion of additional conditional branches following the same validation criteria described above. We leave it as a tunable parameter in our evaluation (referred to as *interval size*), given that its optimal setting depends on the input.

7 Evaluation Methodology

Experimental Setup. We implement our source-to-source compiler, BitGen in Python that generates optimized GPU kernels from bitstream programs produced by Parabix [24, 50]. The CUDA kernels are compiled at runtime using NVRTC [6]. During execution, the GPU first launches a preprocessing kernel to transpose the input data into bitstreams (Section 2). The generated kernel then processes the bitstream instructions for regex matching, including character classes and other patterns. Experiments are conducted primarily on an NVIDIA RTX 3090 (Ampere, 24 GB, 82 SMs), with evaluations on an H100 NVL (Hopper, 94 GB, 132 SMs) and L40S (Ada, 48 GB, 142 SMs) to assess portability. CPU schemes are run on an Intel Xeon Platinum 8562Y+ (32 cores, hyperthreading disabled). All programs are compiled with CUDA 12.4 and GCC 13.2. We use architecture-specific CUDA compilation flags to match the native GPU architecture of the target device.

Each benchmark is executed 10 times, and we report the average execution time, including both the input transpose and the generated kernel for bitstream processing. The transpose on input is highly parallel and efficient; transposing 1 MB on an RTX 3090 typically takes about 0.026 ms (37,449 MB/s), regardless of the regex patterns or input data, causing negligible performance overhead. Following prior work [37, 39], we report steady-state kernel execution time for each benchmark using its predefined regex patterns. Compilation and data transfer time are excluded, as they can be amortized over multiple runs or overlapped during execution. Throughput is measured in MB/s as the number of input symbols

Table 1: Statistics of the evaluated applications, including the number of regular expressions, their average (Avg.) and standard deviation (SD.) in character length, and the instruction count breakdown of their corresponding bitstream programs.

App	#Regex	Regex Length		#Instruction				
		Avg.	SD.	and	or	not	shift	while
Brill	1,849	44.4	16.9	82,604	21,227	19,124	48,983	15,028
ClamAV	491	359.7	310.7	71,135	4,469	4,855	45,129	566
Dotstar	1,279	52.8	30.8	68,311	5,600	4,949	42,598	183
Protomata	2,338	96.5	36.2	63,809	44,291	8,772	31,580	305
Snort	1,873	50.5	41.5	84,481	18,608	10,725	47,560	4,742
Yara	3,358	32.5	24.9	105,612	8,332	5,162	76,756	7
Bro217	227	34.1	27.9	8,918	1,025	2,339	2,598	11
ExactMatch	298	52.9	19.2	25,582	1,242	2,945	12,197	2
Ranges1	298	54.3	19.4	27,256	2,263	3,710	12,421	238
TCP	300	53.9	21.4	26,830	1,827	3,363	12,507	149

processed per second. We use Nsight Compute [4] to collect detailed kernel metrics. We validate results by comparing match positions and counts against icgrep's reference output.

Evaluated Schemes. We compare BitGen with three state-of-the-art regex matching systems: (1) *ng*AP [37, 38], a GPU-based engine with non-blocking automata processing optimizations; (2) icgrep (v1.0) [24], a CPU-based SIMD engine built on Parabix and bitstream processing; (3) Hyperscan (v4.4.1) [3, 83], Intel's industry-standard CPU engine featuring NFA decomposition, SIMD acceleration, string matching optimizations, and prefiltering. All systems are configured with recommended optimizations: *ng*AP is tuned per application for best GPU performance. We evaluate Hyperscan in both single-threaded (HS-1T) and multi-threaded (HS-MT) modes [3, 13]. HS-MT parallelizes across regexes, but its scalability varies across applications due to factors such as cache contention, memory bandwidth saturation, and workload imbalance. For each application, we sweep the number of threads (1, 2, 4, 8, 16, 32) and report the best-performing configuration for HS-MT.

Parameter Setup. BitGen exposes four key parameters: two for our proposed optimizations and two for CUDA kernel execution. The optimization parameters are *merge size* and *interval size*. Merge size defines the maximum number of SHIFT instructions merged into one (Section 5.3), while interval size controls the frequency of inserting conditional branches to skip unnecessary computation (Section 6). The CUDA-related parameters are *CTA count* and *max register number*. CTA count sets the number of CTAs for kernel execution. Regexes are partitioned into groups with similar total character length, each assigned to one CTA to balance GPU workload. Max register number limits the registers per thread (via `-maxrregcount`) to reduce spilling while maintaining occupancy. For overall performance evaluation, we use the best-tuned parameters for each application. In other experiments, such as performance breakdowns, we use fixed default values: merge size = 8, interval size = 8, CTA count = 256, and max register number = 128. We also conduct sensitivity studies on the optimization parameters to analyze their impact on performance in Section 8.2.

Benchmarks. We evaluate performance using ten real-world regex applications from benchmark suites: AutomataZoo [82], ANMLZoo [80], and Regex [19]. Since *ng*AP and icgrep lack support for some regex features, we select only regexes supported by all systems. Table 1 summarizes these applications, including regex details and bitstream instruction type breakdowns. We use MNCaRT [13] and VASim [81] to convert regexes into automata for *ng*AP. Each application processes 10^6 bytes of input from the benchmark suites, representing real-world scenarios for practical relevance.

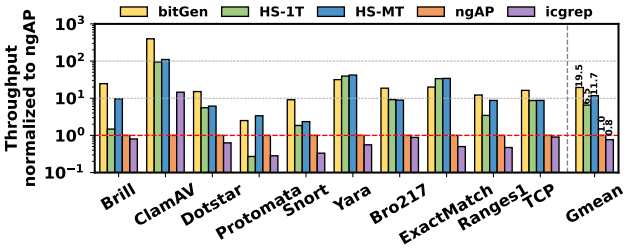


Figure 11: Throughput results normalized to *ng*AP.

Table 2: Throughput of evaluated applications. Thpt: Throughput in MB/s. SpdUp: Speedup of BitGen over each scheme.

App	bitGen	HS-1T		HS-MT		*ng*AP		icgrep	
	Thpt	Thpt	SpdUp	Thpt	SpdUp	Thpt	SpdUp	Thpt	SpdUp
Brill	**85.3**	5.1	16.7×	33.4	2.6×	3.5	24.4×	2.8	30.5×
ClamAV	**1026.8**	244.2	4.2×	284.4	3.6×	2.6	394.9×	37.6	27.3×
Dotstar	**678.9**	249.4	2.7×	275.7	2.5×	44.9	15.1×	28.3	24.0×
Protomata	15.7	1.7	9.2×	**21.1**	0.7×	6.3	2.5×	1.8	8.7×
Snort	**391.8**	79.6	4.9×	101.0	3.9×	43.0	9.1×	14.3	27.4×
Yara	638.3	793.7	0.8×	**847.2**	0.8×	20.2	31.6×	11.3	56.5×
Bro217	**2013.2**	991.8	2.0×	991.8	2.0×	108.2	18.6×	95.5	21.1×
ExactMatch	1986.5	3348.2	0.6×	**3398.7**	0.6×	99.5	20.0×	49.8	39.9×
Ranges1	**1246.1**	352.5	3.5×	891.0	1.4×	102.2	12.2×	48.2	25.9×
TCP	**1678.1**	894.8	1.9×	900.1	1.9×	103.1	16.3×	93.3	18.0×
Gmean	–	–	3.0×	–	1.7×	–	19.5×	–	25.3×

8 Experimental Results

8.1 Overall Performance

Figure 11 shows the normalized throughput of the evaluated applications. BitGen significantly outperforms both CPU and GPU baselines, achieving an average speedup of 3.0× over single-threaded Hyperscan, 1.7× over multi-threaded Hyperscan, 19.5× over *ng*AP, and 25.3× over icgrep. The absolute throughput of those applications is shown in Table 2.

***Comparison to* ngAP.** BitGen consistently outperforms *ng*AP across all applications, achieving up to 394.9× speedup on `ClamAV`, with more than 10× improvement in 8 out of 10 applications. These gains result from BitGen's efficient bit-parallel approach to regex matching. In contrast, *ng*AP relies on an NFA-based method that requires multiple memory accesses for each symbol-state comparison, leading to irregular memory patterns. Although it adopts a worklist design to expose symbol-level parallelism, the added memory overhead and limited worklist size can restrict GPU utilization. For instance, `ClamAV` includes many regex patterns for virus byte sequences. For most non-virus inputs, only a small number of states become active, resulting in short worklists that fail to saturate GPU

Table 3: Breakdown of BitGen to evaluate effect of each optimization.

Abbr.	Dependency-Aware Thread-Data Mapping		Shift Rebalancing	Zero Block Skipping
	Static	Dynamic		
Base				
DTM–	✓			
DTM	✓	✓		
SR	✓	✓	✓	
ZBS	✓	✓	✓	✓

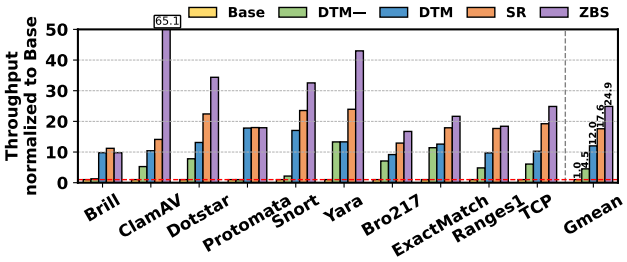


Figure 12: Performance breakdown of BitGen.

resources. BitGen avoids this issue by using compact bitwise operations. Our interleaved execution model allows most computations to happen in registers, reducing memory accesses and control flows.

Comparison to Hyperscan. BitGen achieves a geometric mean speedup of 3.0× and 1.7× over HS-1T and HS-MT, respectively. In particular, it achieves up to 3.9× improvement on `Snort` compared to HS-MT. HS-MT outperforms BitGen only on three applications (`Protomata`, `Yara`, and `ExactMatch`). We observe that most regexes in these applications are simple string patterns, which benefit from Hyperscan's decomposition optimizations and highly-tuned SIMD-based string matching implementation [61].

HS-MT exhibits *limited scalability*: its multithreaded performance is only 1.76× that of HS-1T, constrained by cache contention, memory bandwidth saturation, and workload imbalance. BitGen achieves an average 1.7× speedup over HS-MT. While RTX 3090 has ~4.5× higher theoretical integer throughput than the Xeon 8562Y+ (17.8 TIOPS vs. 3.9 TIOPS), this ratio does not directly translate to application-level speedup because Hyperscan incorporates many specialized optimizations, particularly for string literals, that significantly reduce the amount of computation on the CPU [61]. In contrast, *BitGen focuses solely on bitstream programs and does not include regex-specific optimizations; these are orthogonal to our approach and could be incorporated to further improve performance in future work.*

Comparison to icgrep. Although both BitGen and icgrep use bitstream processing for regular expression matching, BitGen consistently delivers much higher performance–achieving an average speedup of 25.3× across all applications. This performance gap results from BitGen's effective utilization of GPU's massive parallelism on bitstreams. The performance difference is particularly evident in large-scale applications such as `Brill`, `Yara`, and `Snort`, where icgrep's execution efficiency degrades considerably.

Discussion: Applicability to CPUs. The core optimizations in BitGen are specifically designed for GPU architectures and do not directly apply to CPUs. First, CPUs rely on narrow SIMD units (e.g., AVX-512) that operate in lockstep and require uniform control flow. BitGen's dynamic thread-to-data mapping and zero block skipping break this uniformity and is inefficient on CPU SIMD lanes. Second, BitGen leverages shared memory for intra-thread communication, which has no counterpart on CPUs. Finally, BitGen tolerates redundant recomputation to resolve cross-block dependencies, a tradeoff that is acceptable on GPUs but too costly on CPUs with limited parallelism.

8.2 Optimization Analysis

To evaluate the impact of each optimization in BitGen, we compare its performance with and without each technique, followed by a profiling analysis under varying parameters.

Performance Breakdown. To understand the impact of each optimization, we add them one at a time (as shown in Table 3) and report performance after each step. The baseline only fuses loops with bitwise operations, resulting in a sequential block-wise execution with partial interleaving. We exclude sequential execution from our baseline comparison because it stores all intermediate bitstreams, leading to excessive memory usage that exceeds the available GPU memory. Figure 12 shows BitGen's performance normalized to the baseline. Dependency-Aware Thread-Data Mapping is split into `DTM-` (static analysis only) and `DTM` (with dynamic analysis). `DTM-` improves shift-heavy applications like `Yara` (13.2×), while `DTM` further benefits control-intensive ones like `Brill` (9.8×) and `Protomata` (17.8×). Adding Shift Rebalancing (SR) raises performance to 17.6× over the baseline. It is especially effective for bitstream programs with long dependency chains, such as `ExactMatch` and `ClamAV`, which gain 1.4× and 1.3× over DTM alone. Zero Block Skipping (ZBS) further increases the average speedup to 24.9×. For example, `Dotstar` improves to 34.4× due to frequent zero-block sequences in its regex patterns.

Memory Consumption and DRAM Accesses with DTM. Table 4 shows the profiling results about memory consumption and DRAM accesses for different levels of fusion in DTM. On average across all evaluated applications, *each CTA* in the baseline, when processing 1 MB of input, performs hundreds of MBs of DRAM accesses. For example, `Brill` incurs ~102.2 GB of total DRAM traffic across all CTAs. The large size of intermediate bitstreams leads to excessive memory accesses and can exceed GPU memory capacity when processing larger inputs, significantly reducing scalability. In contrast, DTM merges all bitstream instructions into one loop, thus does not need intermediate bitstreams. At runtime, it minimizes memory traffic, issuing only 0.2 MB of DRAM reads and writes per CTA on average.

Recompute Overhead of DTM. Table 5 summarizes the recomputation overhead of Dependency-Aware Thread-Data Mapping. For most applications, only a small number of bits need to be recomputed per iteration. Even in applications with complex control flow like Brill and Protomata, the overhead is limited to approximately one additional iteration of computation, demonstrating DTM's efficiency across diverse regex workloads.

Table 4: Compile-time and runtime profiling results in Dependency-Aware Thread-Data Mapping, reported as the average per CTA across all evaluated applications.

Scheme	Compile-time		Runtime	
	#Loop	#Intermediate Bitstream	DRAM Read (MB)	DRAM Written (MB)
Base	260.7	317.8	177.9	85.2
DTM–	17.6	54.2	124.4	53.6
DTM	1	0	0.2	0.2

Table 5: Recomputation Overhead in Dependency-Aware Thread-Data Mapping. DTM recomputes both static and dynamic overlap distances.

App	Overlap Distance (bit)			Recompute %	#Iter
	Avg. Static	Avg. Dynamic	Max Dynamic		
Brill	3.2	160.1	514	1.00	63.1
ClamAV	2.9	0.1	209	< 0.01	62.2
Dotstar	2.8	0.7	72	< 0.01	62.0
Protomata	2.1	346.3	11678	2.13	63.4
Snort	3.2	2.5	489	< 0.01	62.2
Yara	5.0	< 0.1	8	< 0.01	63.0
Bro217	0.2	0	0	< 0.01	62.0
ExactMatch	0.8	< 0.1	2	< 0.01	62.0
Ranges1	0.8	0.9	24	< 0.01	62.0
TCP	0.8	0.1	30	< 0.01	62.0

Discussion: Limits of Overlap Distance in DTM. In Dependency-Aware Thread-Data Mapping, the number of dependent bits to recompute is determined by the accumulated overlap distance. This distance can be dynamic when loops are present and may exceed the size of a block. When this happens, dependency bits beyond the block boundary become inaccessible, potentially leading to incorrect or incomplete pattern matching. For example, specific patterns such as `/.*/`, which match any sequence of characters except line breaks, can exceed the overlap limit when processing very long single-line inputs. With 512 threads and 32 bits per thread, the maximum overlap distance is 16,384 bits (i.e., approximately 16 KB of input). If the required overlap distance exceeds this limit, the current block would depend on multiple previous blocks, which cannot be handled by interleaved execution. As shown in Table 5, our evaluated applications do not encounter this issue. To address this limitation, one possible solution is to introduce a fallback mechanism. If a loop reaches the overlap limit during execution, threads within the CTA could fall back to sequential execution to generate an intermediate bitstream for this loop. Subsequent interleaved execution could then skip the loop and directly consume this bitstream. We leave the fallback mechanism as future work.

Sensitivity to Merge Size and Profiling Results in SR. We evaluate the impact of merge size in SR using four configurations: 1, 4, 16, and 32. Figure 13 shows that performance improves with larger merge sizes, as more shift instructions can be merged. However, a very large merge size can demand too much shared memory. Table 6 shows that larger merge sizes reduce barriers, lower percentage of stall cycles due to barriers, and merge more shared memory stores, resulting in fewer memory accesses overall.

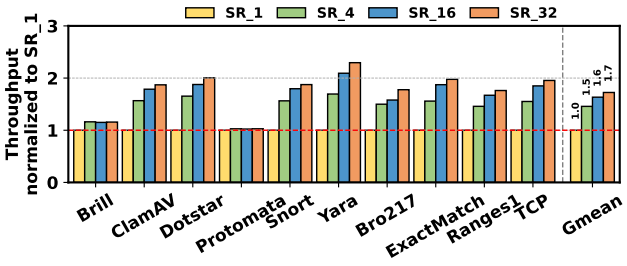


Figure 13: Normalized throughput of Shift Rebalancing with different shared memory store merge sizes (1, 4, 16, 32).

Table 6: Compile-time and runtime profiling results of Shift Rebalancing for various merge sizes, reported as the average per CTA across all evaluated applications. #Sync: Number of barriers in `SHIFT` instructions. SMem: Shared memory.

Scheme	Compile-time		Runtime	
	#Sync	SMem Size (KB)	Barrier Stall %	SMem Access (MB)
SR_1	305.1	2	49.6	70.2
SR_4	87.2	8	27.4	67.9
SR_16	41.4	32	19.0	63.9
SR_32	35.3	64	17.5	61.4

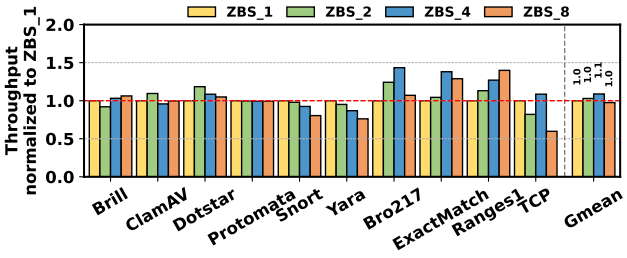


Figure 14: Normalized throughput of Zero Block Skipping under different interval sizes (1, 2, 4, 8).

***Sensitivity to Interval Size in* ZBS.** We evaluate ZBS with interval sizes of 1, 2, 4, and 8. When the interval size is 1, the zero-path guard is inserted at every check point, maximizing skips but increasing branching and synchronization overhead. Figure 14 demonstrates the results. We observe that the optimal size varies by application, as it depends on both input data and regex patterns. For example, TCP performs best with interval size 4. Generally, a size of 1 can add too many branches and sync overhead, while size 8 may miss chances to skip redundant instructions. Figure 12 shows that enabling ZBS with the default interval size of 8 increases the speedup from 17.6 to 24.9 compared to no ZBS.

8.3 Portability Studies

We evaluate BitGen's performance portability across GPU architectures by comparing its throughput on the NVIDIA H100 and

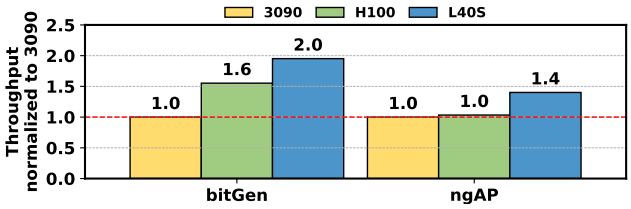


Figure 15: Throughput of BitGen and *ng*AP on different GPUs, normalized to RTX 3090.

L40S, normalized to the baseline RTX 3090. Figure 15 shows that BitGen achieves 1.6× and 2.0× speedups on H100 and L40S, respectively, while *ng*AP shows no improvement on H100 and only 1.4× on L40S. This difference is due to BitGen becoming *compute-bound* after applying interleaved execution. Unlike *ng*AP, which is more latency-sensitive and less parallel, BitGen maps well to massive thread-level parallelism and bitwise compute. The observed speedup of BitGen aligns closely with the theoretical integer compute throughput across these GPUs: 17.8, 33.5, and 45.8 TIOPS for 3090, H100, and L40S (≈ 1:1.9:2.6). Despite H100's higher memory bandwidth (HBM3 vs. GDDR6 on L40S), BitGen achieves better performance on L40S due to its higher integer compute capacity from more CUDA cores.

9 Related Work

The performance demands of regex matching have motivated extensive research on hardware accelerators [22, 25, 31, 33, 41, 47, 51, 56, 57, 66–74, 79, 85, 94]. This section focuses instead on execution engines for CPUs and GPUs, and compares them with BitGen.

Regex Engines. Traditional regex engines often rely on backtracking, which can lead to exponential runtime in the worst case [1, 16, 29, 30]. RE2 avoids this by compiling regexes into DFAs, ensuring linear-time performance [5]. Prior work accelerates regex matching by parallelizing automata via prefix sum, path enumeration, or speculation [43, 54, 60, 87, 92, 93]. These optimizations are orthogonal to our approach, as they aim to improve parallelism. However, their efficiency drops when handling large sets of patterns due to increased work [53]. This limitation is critical in applications like intrusion detection [7, 64, 91], antivirus scanning [2, 8], and bioinformatics [21, 65], which require matching thousands of regexes. Since our work targets this multi-regex setting, we assign each regex group to a single CTA to simplify synchronization using intra-CTA barriers at the cost of per-regex scalability. Nonetheless, bitstream programs are naturally data-parallel, and our techniques can be extended to multi-CTA execution with more complex synchronization, which we leave to future work.

While some engines [45, 74] adopt variants of the Aho-Corasick algorithm [9] to support multi-string matching, these methods do not generalize to regexes. Hyperscan [83] addresses multi-regex workloads by a hybrid approach: it decomposes complex patterns into simpler components, applies fast string matching where possible, and leverages Glushkov NFAs [40] for the complex regexes. Prior works [26, 52, 53, 76–78, 90, 95] have explored automata-based engines for multi-regex matching on GPUs, but their performance is often limited by irregular memory access patterns and divergent control flows. Avalle *et al.* [14] propose a multi-striding transformation for NFAs to enable processing of multiple bytes simultaneously; however, the transformed NFAs incur a significant increase in size. *ng*AP [37, 38] improves the compute utilization by non-blocking execution, however, its memoization table requires large memory footprint, limiting scalability. In contrast, BitGen targets GPU-based multi-pattern matching by leveraging bit-parallelism and optimizing for resolving dependency and data reuse, enabling high-throughput execution of large-scale regex workloads.

Bit-Parallel Pattern Matching. Bit-parallel algorithms such as Shift-And and its variants [15, 86] achieve efficient string matching via bitwise operations. Recent works extend this idea to general regular expressions by simulating NFAs with bit-parallelism [39, 48], often combined with regex-level rewrites for further optimization. However, these bit-parallel NFA algorithms differ fundamentally from our approach: they represent active NFA states as bit vectors and process the input one byte at a time, performing parallel state transitions using bitwise logic. In contrast, our method operates directly on bitstreams from input text and matches multiple characters per step. Building on bitstream representations, Parabix [24, 50] compiles regular expressions into bitstream programs that process input in parallel using SIMD-style operations. Qiu *et al.* [59] further improve the parallelism of bitstream programs by speculating on control and data dependencies involving global states. However, their approach targets CPU execution and does not consider the high-throughput challenges of executing multiple regex patterns in parallel on GPUs. In contrast, while Qiu *et al.* [59] focus on data dependencies arising from global loop states (e.g., whether the current iteration is odd or even), our work targets data dependencies across bitstream blocks, which we resolve efficiently using Dependency-Aware Thread-Data Mapping.

10 Conclusions

We present BitGen, a GPU-targeted code generator that enables efficient interleaved execution of bitstream programs, a powerful abstraction widely used in unstructured data analytics tasks such as regex matching. We identify key limitations of sequential block-wise execution on GPUs, including poor data reuse, excessive memory consumption, and redundant computation. To address these issues, BitGen enables interleaved execution and introduces GPU-specific optimizations to make it practical and efficient. Experiments on real-world regex workloads show that BitGen significantly outperforms state-of-the-art GPU and CPU regex engines.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback, which significantly improved the quality of this paper, and Ming Li for proofreading. Tianao Ge and Xiaowen Chu were partially supported by the National Natural Science Foundation of China under Grant No. 62302416 and the Guangzhou Municipal Joint Funding Project with Universities and Enterprises under Grant No. 2024A03J0616. Hongyuan Liu acknowledges the High Performance Computing Cluster at Stevens Institute of Technology and the Stevens Institute for Artificial Intelligence for providing part of the computing resources. Corresponding authors: Hongyuan Liu and Xiaowen Chu.

References

[1] 2025. Boost Regex Library. https://github.com/boostorg/regex.

[2] 2025. Clamav net. https://www.clamav.net/.

[3] 2025. HSCompile: MNRL HyperScan. https://github.com/kevinaangstadt/hscompile.

[4] 2025. NVIDIA Nsight Compute Profiling Tool. https://docs.nvidia.com/nsight-compute/NsightCompute/.

[5] 2025. RE2. https://github.com/google/re2.

[6] 2025. The User guide for the NVRTC library. https://docs.nvidia.com/cuda/nvrtc/.

[7] 2025. The Zeek Network Security Monitor. https://www.zeek.org.

[8] 2025. YARA: The pattern matching swiss knife for malware researchers. https://virustotal.github.io/yara/.

[9] Alfred V. Aho and Margaret J. Corasick. 1975. Efficient string matching: an aid to bibliographic search. *Commun. ACM* (1975). https://doi.org/10.1145/360825.360855

[10] V Aho Alfred, S Lam Monica, and D Ullman Jeffrey. 2007. *Compilers Principles, Techniques & Tools*. Pearson Education.

[11] Mohammed Alser, Julien Eudine, and Onur Mutlu. 2025. Taming Large-scale Genomic Analyses via Sparsified Genomics. *Nature Communications* 16, 1 (2025), 876. https://doi.org/10.1038/s41467-024-55762-1

[12] Mehmet Altınel and Michael J Franklin. 2000. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB)*. 53–64.

[13] K. Angstadt, J. Wadden, V. Dang, T. Xie, D. Kramp, W. Weimer, M. Stan, and K. Skadron. 2018. MNCaRT: An Open-Source, Multi-Architecture Automata-Processing Research and Execution Ecosystem. *IEEE Computer Architecture Letters (CAL)* (2018).

[14] Matteo Avalle, Fulvio Risso, and Riccardo Sisto. 2016. Scalable Algorithms for NFA Multi-Striding and NFA-Based Deep Packet Inspection on GPUs. *IEEE/ACM Transactions on Networking (ToN)* (2016). https://doi.org/10.1109/TNET.2015.2429918

[15] Ricardo Baeza-Yates and Gaston H. Gonnet. 1992. A New Approach to Text Searching. *Commun. ACM* 35, 10 (1992). https://doi.org/10.1145/135239.135243

[16] Efe Barlas, Xin Du, and James C. Davis. 2022. Exploiting Input Sanitization for Regex Denial of Service. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*. https://doi.org/10.1145/3510003.3510047

[17] Michela Becchi and Patrick Crowley. 2007. A Hybrid Finite Automaton for Practical Deep Packet Inspection. In *Proceedings of the 2007 ACM CoNEXT Conference (CoNEXT)*.

[18] Michela Becchi and Patrick Crowley. 2008. Extending finite automata to efficiently match Perl-compatible regular expressions. In *Proceedings of the 2008 ACM CoNEXT Conference*. https://doi.org/10.1145/1544012.1544037

[19] Michela Becchi, Mark Franklin, and Patrick Crowley. 2008. A Workload for Evaluating Deep Packet Inspection Architectures. In *Proceedings of the International Symposium on Workload Characterization (IISWC)*.

[20] Amit Bleiweiss and Nicole Luo. 2024. Mastering LLM Techniques: Data Preprocessing. https://developer.nvidia.com/blog/mastering-llm-techniques-data-preprocessing/.

[21] Chunkun Bo, Vinh Dang, Elaheh Sadredini, and Kevin Skadron. 2018. Searching for Potential gRNA Off-Target Sites for CRISPR/Cas9 Using Automata Processing Across Different Platforms. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*. https://doi.org/10.1109/HPCA.2018.00068

[22] Benjamin C. Brodie, David E. Taylor, and Ron K. Cytron. 2006. A Scalable Architecture For High-Throughput Regular-Expression Pattern Matching. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA)*. https://doi.org/10.1109/ISCA.2006.7

[23] Damla Senol Cali, Gurpreet S. Kalsi, Zülal Bingöl, Can Firtina, Lavanya Subramanian, Jeremie S. Kim, Rachata Ausavarungnirun, Mohammed Alser, Juan Gomez-Luna, Amirali Boroumand, Anant Norion, Allison Scibisz, Sreenivas Subramoneyon, Can Alkan, Saugata Ghose, and Onur Mutlu. 2020. GenASM: A High-Performance, Low-Power Approximate String Matching Acceleration Framework for Genome Sequence Analysis. In *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. https://doi.org/10.1109/MICRO50266.2020.00081

[24] Robert D. Cameron, Thomas C. Shermer, Arrvindh Shriraman, Kenneth S. Herdy, Dan Lin, Benjamin R. Hull, and Meng Lin. 2014. Bitwise data parallelism in regular expression matching. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT)*. 139–150. https://doi.org/10.1145/2628071.2628079

[25] Filippo Carloni, Davide Conficconi, and Marco D. Santambrogio. 2024. ALVEARE: a Domain-Specific Framework for Regular Expressions. In *Proceedings of the 61st ACM/IEEE Design Automation Conference (DAC)*. https://doi.org/10.1145/3649329.3657378

[26] Niccolo' Cascarano, Pierluigi Rolando, Fulvio Risso, and Riccardo Sisto. 2010. iNFAnt: NFA Pattern Matching on GPGPU Devices. *SIGCOMM Computer Communication Review (CCR)* (2010).

[27] Qianxi Chen, Yujiao Deng, Qiang Wu, and Zhixiong Di. 2025. An r-DFA-based Layout Pattern Match Method Supporting Fuzzy Matching. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2025). https://doi.org/10.1109/TCAD.2025.3556969

[28] Luisa Cicolini, Filippo Carloni, Marco D. Santambrogio, and Davide Conficconi. 2024. One Automaton to Rule Them All: Beyond Multiple Regular Expressions Execution. In *Proceedings of the 2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. https://doi.org/10.1109/CGO57630.2024.10444810

[29] Russ Cox. 2007. Regular Expression Matching Can Be Simple and Fast. https://swtch.com/~rsc/regexp/regexp1.html.

[30] Scott A Crosby and Dan S Wallach. 2003. Denial of service via algorithmic complexity attacks. In *Proceedings of the 12th USENIX Security Symposium (USENIX Security)*.

[31] Paul Dlugosch, Dave Brown, Paul Glendenning, Michael Leventhal, and Harold Noyes. 2014. An Efficient and Scalable Semiconductor Architecture for Parallel Automata Processing. *IEEE Transactions on Parallel and Distributed Systems* (2014).

[32] Johannes Doleschal, Benny Kimelfeld, and Wim Martens. 2021. Database Principles and Challenges in Text Analysis. *SIGMOD Rec.* (2021). https://doi.org/10.1145/3484622.3484624

[33] Xingran Du, Joel S. Emer, and Daniel Sanchez. 2025. Hopps: Leveraging Sparsity to Accelerate Automata Processing. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 96–111. https://doi.org/10.1145/3676642.3736126

[34] Yuanwei Fang, Tung T. Hoang, Michela Becchi, and Andrew A. Chien. 2015. Fast support for unstructured data processing: the unified automata processor. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*. https://doi.org/10.1145/2830772.2830809

[35] Yuanwei Fang, Chen Zou, and Andrew A. Chien. 2019. Accelerating raw data analysis with the ACCORDA software and hardware architecture. *Proc. VLDB Endow.* (2019). https://doi.org/10.14778/3342263.3342634

[36] Fernando Florenzano, Cristian Riveros, Martín Ugarte, Stijn Vansummeren, and Domagoj Vrgoč. 2020. Efficient Enumeration Algorithms for Regular Document Spanners. *ACM Trans. Database Syst.* (2020). https://doi.org/10.1145/3351451

[37] Tianao Ge, Tong Zhang, and Hongyuan Liu. 2024. ngAP: Non-blocking Large-scale Automata Processing on GPUs. In *Proceedings of the ACM International Conference on International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/3617232.3624848

[38] Tianao Ge, Tong Zhang, and Hongyuan Liu. 2025. Towards Scalable and Non-blocking Automata Processing on GPUs with ngAP. *ACM Trans. Comput. Syst.* (2025). https://doi.org/10.1145/3748646

[39] Alexis Le Glaunec, Lingkun Kong, and Konstantinos Mamouras. 2024. HybridSA: GPU Acceleration of Multi-pattern Regex Matching Using Bit Parallelism. *Proc. ACM Program. Lang.* 8, OOPSLA2 (2024), 1699–1728. https://doi.org/10.1145/3689771

[40] Victor Mikhaylovich Glushkov. 1961. The Abstract Theory of Automata. *Russian Mathematical Surveys* 16, 5 (1961), 1.

[41] Vaibhav Gogte, Aasheesh Kolli, Michael J. Cafarella, Loris D'Antoni, and Thomas F. Wenisch. 2016. HARE: Hardware accelerator for regular expressions. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. https://doi.org/10.1109/MICRO.2016.7783747

[42] Robert Heeg. 2023. Possibilities and limitations, of unstructured data. https://researchworld.com/articles/possibilities-and-limitations-of-unstructured-data.

[43] W. Daniel Hillis and Guy L. Steele. 1986. Data Parallel Algorithms. *Commun. ACM* (Dec. 1986), 1170–1183. https://doi.org/10.1145/7902.7903

[44] Yi Huang, Lingkun Kong, Dibei Chen, Zhiyu Chen, Xiangyu Kong, Jianfeng Zhu, Konstantinos Mamouras, Shaojun Wei, Kaiyuan Yang, and Leibo Liu. 2023. CASA: An Energy-Efficient and High-Speed CAM-based SMEM Seeding Accelerator for Genome Alignment. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. https://doi.org/10.1145/3613424.3614313

[45] Theo Jepsen, Daniel Alvarez, Nate Foster, Changhoon Kim, Jeongkeun Lee, Masoud Moshref, and Robert Soulé. 2019. Fast String Searching on PISA. In *Proceedings of the 2019 ACM Symposium on SDN Research*. https://doi.org/10.1145/3314148.3314356

[46] Seongyoung Kang, Jiyoung An, Jinpyo Kim, and Sang-Woo Jun. 2021. MithriLog: Near-Storage Accelerator for High-Performance Log Analytics. In *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[47] Lingkun Kong, Qixuan Yu, Agnishom Chattopadhyay, Alexis Le Glaunec, Yi Huang, Konstantinos Mamouras, and Kaiyuan Yang. 2022. Software-Hardware Codesign for Efficient in-Memory Regular Pattern Matching. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*.

[48] Alexis Le Glaunec, Lingkun Kong, and Konstantinos Mamouras. 2023. Regular Expression Matching using Bit Vector Automata. *Proc. ACM Program. Lang.* 7,

OOPSLA1 (2023). https://doi.org/10.1145/3586044

[49] Yinan Li, Nikos R. Katsipoulakis, Badrish Chandramouli, Jonathan Goldstein, and Donald Kossmann. 2017. Mison: A Fast JSON Parser for Data Analytics. *Proceedings of the VLDB Endowment (VLDB)* 10, 10 (2017), 1118–1129. https://doi.org/10.14778/3115404.3115416

[50] Dan Lin, Nigel Medforth, Kenneth S. Herdy, Arrvindh Shriraman, and Robert D. Cameron. 2012. Parabix: Boosting the efficiency of text processing on commodity processors. In *Proceedings of the 18th IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 373–384. https://doi.org/10.1109/HPCA.2012.6169041

[51] Hongyuan Liu, Mohamed Ibrahim, Onur Kayiran, Sreepathi Pai, and Adwait Jog. 2018. Architectural Support for Efficient Large-Scale Automata Processing. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*.

[52] Hongyuan Liu, Sreepathi Pai, and Adwait Jog. 2020. Why GPUs are Slow at Executing NFAs and How to Make them Faster. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 251–265. https://doi.org/10.1145/3373376.3378471

[53] Hongyuan Liu, Sreepathi Pai, and Adwait Jog. 2023. Asynchronous Automata Processing on GPUs. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 7, 1 (2023). https://doi.org/10.1145/3579453

[54] Todd Mytkowicz, Madanlal Musuvathi, and Wolfram Schulte. 2014. Data-parallel Finite-state Machines. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[55] Marziyeh Nourian, Xiang Wang, Xiaodong Yu, Wu-chun Feng, and Michela Becchi. 2017. Demystifying automata processing: GPUs, FPGAs or Micron's AP?. In *Proceedings of the International Conference on Supercomputing (ICS)*. https://doi.org/10.1145/3079079.3079100

[56] Taejune Park, Jaehyun Nam, Seung Ho Na, Jaewoong Chung, and Seungwon Shin. 2021. Reinhardt: Real-time Reconfigurable Hardware Architecture for Regular Expression Matching in DPI. In *Proceedings of the 37th Annual Computer Security Applications Conference (ACSAC)*. https://doi.org/10.1145/3485832.3485878

[57] Daniele Parravicini, Davide Conficconi, Emanuele Del Sozzo, Christian Pilato, and Marco D. Santambrogio. 2021. CICERO: A Domain-Specific Architecture for Efficient Regular Expression Matching. *ACM Trans. Embed. Comput. Syst.* (2021). https://doi.org/10.1145/3476982

[58] Junqiao Qiu and Ali Ebnenasir. 2023. Exploring Scalable Parallelization for Edit Distance-Based Motif Search. *IEEE/ACM Transactions on Computational Biology and Bioinformatics* (2023). https://doi.org/10.1109/TCBB.2022.3208867

[59] Junqiao Qiu, Lin Jiang, and Zhijia Zhao. 2020. Challenging Sequential Bitstream Processing via Principled Bitwise Speculation. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/3373376.3378461

[60] Junqiao Qiu, Xiaofan Sun, Amir Hossein Nodehi Sabet, and Zhijia Zhao. 2021. Scalable FSM Parallelization via Path Fusion and Higher-Order Speculation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[61] Kun Qiu, Harry Chang, Yang Hong, Wenjun Zhu, Xiang Wang, and Baoqian Li. 2021. Teddy: An Efficient SIMD-based Literal Matching Engine for Scalable Deep Packet Inspection. In *Proceedings of the 50th International Conference on Parallel Processing (ICPP)*. https://doi.org/10.1145/3472456.3473512

[62] Cristian Riveros, Nicolás Van Sint Jan, and Domagoj Vrgoč. 2023. REmatch: A Novel Regex Engine for Finding All Matches. *Proc. VLDB Endow.* (2023). https://doi.org/10.14778/3611479.3611488

[63] Matt Robinson. 2025. Preprocessing Unstructured Data for LLM Applications. https://www.deeplearning.ai/short-courses/preprocessing-unstructured-data-for-llm-applications/.

[64] Martin Roesch. 1999. Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of the USENIX Conference on System Administration (LISA)*.

[65] Indranil Roy and Srinivas Aluru. 2016. Discovering Motifs in Biological Sequences Using the Micron Automata Processor. *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 13, 1 (2016), 99–111. https://doi.org/10.1109/TCBB.2015.2430313

[66] Elaheh Sadredini, Reza Rahimi, Marzieh Lenjani, Mircea Stan, and Kevin Skadron. 2020. FlexAmata: A Universal and Efficient Adaption of Applications to Spatial Automata Processing Accelerators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/3373376.3378459

[67] Elaheh Sadredini, Reza Rahimi, Marzieh Lenjani, Mircea Stan, and Kevin Skadron. 2020. Impala: Algorithm/Architecture Co-Design for In-Memory Multi-Stride Pattern Matching. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*. https://doi.org/10.1109/HPCA47549.2020.00017

[68] Elaheh Sadredini, Reza Rahimi, Vaibhav Verma, Mircea Stan, and Kevin Skadron. 2019. eAP: A Scalable and Efficient in Memory Accelerator for Automata Processing. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*.

[69] Reetinder Sidhu and Viktor K Prasanna. 2001. Fast Regular Expression Matching Using FPGAs. In *Proceedings of the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 227–238.

[70] David Sidler, Zsolt István, Muhsen Owaida, and Gustavo Alonso. 2017. Accelerating Pattern Matching Queries in Hybrid CPU-FPGA Architectures. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD)*. 403–415.

[71] Andrea Somaini, Filippo Carloni, Giovanni Agosta, Marco D. Santambrogio, and Davide Conficconi. 2025. Combining MLIR Dialects with Domain-Specific Architecture for Efficient Regular Expression Matching. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization (CGO)*. 255–270. https://doi.org/10.1145/3696443.3708916

[72] Arun Subramaniyan and Reetuparna Das. 2017. Parallel Automata Processor. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*. https://doi.org/10.1145/3079856.3080207

[73] Arun Subramaniyan, Jingcheng Wang, Ezhil R. M. Balasubramanian, David Blaauw, Dennis Sylvester, and Reetuparna Das. 2017. Cache Automaton. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*.

[74] Prateek Tandon, Faissal M. Sleiman, Michael J. Cafarella, and Thomas F. Wenisch. 2016. HAWK: Hardware Support for Unstructured Log Processing. In *Proceedings of the IEEE 32nd International Conference on Data Engineering (ICDE)*. 469–480. https://doi.org/10.1109/ICDE.2016.7498263

[75] Ken Thompson. 1968. Programming Techniques: Regular expression search algorithm. *Commun. ACM* (1968). https://doi.org/10.1145/363347.363387

[76] Giorgos Vasiliadis, Spiros Antonatos, Michalis Polychronakis, Evangelos P. Markatos, and Sotiris Ioannidis. 2008. Gnort: High Performance Network Intrusion Detection Using Graphics Processors. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)*.

[77] Giorgos Vasiliadis, Michalis Polychronakis, Spiros Antonatos, Evangelos P Markatos, and Sotiris Ioannidis. 2009. Regular Expression Matching on Graphics Hardware for Intrusion Detection. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)*.

[78] Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. 2011. Parallelization and characterization of pattern matching using GPUs. In *Proceedings of the International Symposium on Workload Characterization (IISWC)*.

[79] Jack Wadden, Kevin Angstadt, and Kevin Skadron. 2018. Characterizing and Mitigating Output Reporting Bottlenecks in Spatial Automata Processing Architectures. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*.

[80] Jack Wadden, Vinh Dang, Nathan Brunelle, Tommy Tracy II, Deyuan Guo, Elaheh Sadredini, Ke Wang, Chunkun Bo, Gabriel Robins, Mircea Stan, and Kevin Skadron. 2016. ANMLzoo: a benchmark suite for exploring bottlenecks in automata processing engines and architectures. In *2016 IEEE International Symposium on Workload Characterization, IISWC 2016, Providence, RI, USA, September 25-27, 2016*. IEEE Computer Society, 105–166. https://doi.org/10.1109/IISWC.2016.7581271

[81] Jack Wadden and Kevin Skadron. 2016. *VASim: An Open Virtual Automata Simulator for Automata Processing Application and Architecture Research.* Technical Report CS2016-03. University of Virginia.

[82] Jack Wadden, Tom Tracy II, Elaheh Sadredini, Lingzi Wu, Chunkun Bo, Jesse Du, Yizhou Wei, Matthew Wallace, Jeffrey Udall, Mircea Stan, and Kevin Skadron. 2018. AutomataZoo: A Modern Automata Processing Benchmark Suite. In *Proceedings of the International Symposium on Workload Characterization (IISWC)*.

[83] Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. 2019. Hyperscan: A Fast Multi-pattern Regex Matcher for Modern CPUs. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. https://doi.org/10.5555/3323234.3323286

[84] Bruce William Watson. 1993. A taxonomy of finite automata construction algorithms. (1993).

[85] Ziyuan Wen, Lingkun Kong, Alexis Le Glaunec, Konstantinos Mamouras, and Kaiyuan Yang. 2024. BVAP: Energy and Memory Efficient Automata Processing for Regular Expressions with Bounded Repetitions. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/3620665.3640412

[86] Sun Wu and Udi Manber. 1992. Fast Text Searching: Allowing Errors. *Commun. ACM* (1992). https://doi.org/10.1145/135239.135244

[87] Yang Xia, Peng Jiang, and Gagan Agrawal. 2020. Scaling Out Speculative Execution of Finite-state Machines with Parallel Merge. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. https://doi.org/10.1145/3332466.3374524

[88] Chengcheng Xu, Shuhui Chen, Jinshu Su, S. M. Yiu, and Lucas C. K. Hui. 2016. A Survey on Regular Expression Matching for Deep Packet Inspection: Applications, Algorithms, and Hardware Platforms. *IEEE Communications Surveys & Tutorials* (2016). https://doi.org/10.1109/COMST.2016.2566669

[89] Fang Yu, Zhifeng Chen, Yanlei Diao, T. V. Lakshman, and Randy H. Katz. 2006. Fast and memory-efficient regular expression matching for deep packet inspection. In *Proceedings of the 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems (ANCS)*. https://doi.org/10.1145/1185347.1185360

[90] Xiaodong Yu and Michela Becchi. 2013. GPU Acceleration of Regular Expression Matching for Large Datasets: Exploring the Implementation Space. In *Proceedings of the ACM International Conference on Computing Frontiers (CF)*. https://doi.org/10.1145/2482767.2482791

[91] Zhipeng Zhao, Hugo Sadok, Nirav Atre, James C. Hoe, Vyas Sekar, and Justine Sherry. 2020. Achieving 100Gbps Intrusion Prevention on a Single Server. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[92] Zhijia Zhao and Xipeng Shen. 2015. On-the-Fly Principled Speculation for FSM Parallelization. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[93] Zhijia Zhao, Bo Wu, and Xipeng Shen. 2014. Challenging the "Embarrassingly Sequential": Parallelizing Finite State Machine-based Computations Through Principled Speculation. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[94] Youwei Zhuo, Jinglei Cheng, Qinyi Luo, Jidong Zhai, Yanzhi Wang, Zhongzhi Luan, and Xuehai Qian. 2018. CSE: Parallel Finite State Machines with Convergence Set Enumeration. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. https://doi.org/10.1109/MICRO.2018.00012

[95] Yuan Zu, Ming Yang, Zhonghu Xu, Lin Wang, Xin Tian, Kunyang Peng, and Qunfeng Dong. 2012. GPU-based NFA Implementation for Memory Efficient High Speed Regular Expression Matching. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*. https://doi.org/10.1145/2370036.2145833

A Artifact Appendix

A.1 Abstract

This artifact includes the source code of BitGen along with other baselines. We provide instructions and scripts to build the codebase and reproduce the experimental results presented in Figure 11, Table 2 and Figure 12 of the paper.

A.2 Artifact check-list (meta-information)

- **Algorithm:** BitGen, *ng*AP [37], icgrep [24], Hyperscan [83]
- **Program:** Python, CUDA C++
- **Compilation:** NVCC 12.4, NVRTC, GCC 13, CMake 3.24.1
- **Binary:** Shared libraries and CUDA binaries
- **Run-time environment:** Ubuntu 20.04 with CUDA 12.4
- **Hardware:** x86_64 CPU with host memory larger than 32 GB. CUDA-enabled GPU with device memory larger than 24 GB.
- **Metrics:** Achieved throughput (bytes per second)
- **Output:** CSV files and running logs
- **How much disk space required (approximately)?:** 60 GiB
- **How much time is needed to prepare workflow (approximately)?:** 1 hour
- **How much time is needed to complete experiments (approximately)?:** 6 hours
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** Apache 2.0
- **Archived (provide DOI)?:** 10.5281/zenodo.16664499

A.3 Description

A.3.1 How to access. The artifact is archived on Zenodo and made available on GitHub for any future updates or revisions.

https://github.com/getianao/BitGen
https://doi.org/10.5281/zenodo.16664499

A.3.2 Hardware dependencies. BitGen is expected to run on NVIDIA GPUs with a compute capability of no less than 8.6.

A.3.3 Software dependencies. All experiments are conducted under Ubuntu 20.04. The artifact requires the NVIDIA CUDA driver of version 550.54.15 or later, and CUDA Toolkit version 12.4.1. The artifact was tested using Python 3.10 and GCC 13. *ng*AP relies on TBB 2020.1 and CMake 3.24. Hyperscan relies on GCC 5.3, Boost, Ragel, and NASM.

A.3.4 Datasets. All datasets are from publicly available benchmark suites: AutomataZoo [82], ANMLZoo [80], and Regex [19]. We converted their regular expressions into ANML format [80] using MNCaRT [13] and VASim [81], and into bitstream programs using icgrep. The preprocessed datasets are included in our repository.

A.4 Installation

We recommend setting up the environment using Docker. Follow the steps below to install the BitGen artifact:

```
$ git clone --recursive \
    https://github.com/getianao/BitGen.git
$ cd ngAP && source env.sh
$ ./1_download_benchmark.sh
$ ./2_build_docker.sh
$ ./3_launch_docker.sh
```

Within the Docker container, execute the following command:

```
$ ./4_build_all.sh
```

This script generates the executables and packages for BitGen, *ng*AP, icgrep, and Hyperscan. For each scheme, we provide a Python wrapper under the `scripts` directory. Users can view the usage instructions by passing the `-h` flag.

A.5 Experiment workflow

To reproduce the experiments in this artifact, execute:

```
$ ./5_run_all.sh
```

The full experiment suite typically takes around 6 hours to complete. All resulting CSV files will be saved in the `results/csv` directory, and log files will be stored in the `log` directory.

A.6 Evaluation and expected results

To generate the figures and tables (Figure 11, Table 2 and Figure 12) from the data in the `results/csv` folder, run:

```
$ ./6_plot_all.sh
```

The generated figures and tables will be stored in the `results` folder. For your reference, we have included results collected on an NVIDIA RTX 3090 and an Intel Xeon 4214R, as well as the corresponding figures and tables, in the `results_ref` folder.

A.7 Experiment customization

Users are encouraged to conduct experiments with various parameters or additional applications by modifying the configuration file (under `configs`) or specifying the options manually. For more details, please refer to `README.md`.

A.8 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-and-badging-current
- https://cTuning.org/ae